AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context

Wei Yang*, Xusheng Xiao†, Benjamin Andow‡, Sihan Li*, Tao Xie*, William Enck‡
*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL
†NEC Laboratories America, Princeton, NJ
‡Department of Computer Science, North Carolina State University, Raleigh, NC
*{weiyang3, sihanli2, taoxie}@illinois.edu, †xsxiao@nec-labs.com, ‡{beandow, whenck}@ncsu.edu

***Abstract*—Mobile malware attempts to evade detection during app analysis by mimicking security-sensitive behaviors of benign apps that provide similar functionality (e.g., sending SMS messages), and suppressing their payload to reduce the chance of being observed (e.g., executing only its payload at night). Since current approaches focus their analyses on the types of security-sensitive resources being accessed (e.g., network), these evasive techniques in malware make differentiating between malicious and benign app behaviors a difficult task during app analysis. We propose that the malicious and benign behaviors within apps can be differentiated based on the contexts that trigger security-sensitive behaviors, i.e., the events and conditions that cause the security-sensitive behaviors to occur. In this work, we introduce AppContext, an approach of static program analysis that extracts the contexts of security-sensitive behaviors to assist app analysis in differentiating between malicious and benign behaviors. We implement a prototype of AppContext and evaluate AppContext on 202 malicious apps from various malware datasets, and 633 benign apps from the Google Play Store. AppContext correctly identifies 192 malicious apps with 87.7% precision and 95% recall. Our evaluation results suggest that the maliciousness of a security-sensitive behavior is more closely related to the intention of the behavior (reflected via contexts) than the type of the security-sensitive resources that the behavior accesses.**

I. Introduction

The increasing popularity of smartphones has made them a target for malware. App markets that distribute software (i.e., apps) to smartphones leverage both automated and manual app analyses to detect malware (e.g., Google Bouncer [1] and Apple App store [2]). To improve the effectiveness of app analysis, existing research proposes approaches that extract features from apps and compare those features against predefined sets of signatures or patterns of malicious or privacy-infringing behaviors, such as method calls, permissions, and information flows [3]–[10].

Similar to PC malware, mobile malware has begun taking steps to evade detection by camouflaging as benign apps [11]. For example, an app can hide malicious intentions by using APIs that are appropriate for its expected functionality. As another example, an app may present itself as a messaging app that sends SMS messages when the user clicks the "send" button. However, it also sends SMS messages containing the user's contact information in the background without notifying the user. Since both of these apps use the same SMS APIs, existing automated tools that consider method calls and information flows are unlikely to distinguish between these cases. Notably, the key difference between these two apps is that the malicious app uses the SMS APIs under an *unexpected context*.

A fundamental difference between malicious and benign apps is that their design principles are different. The principles guiding the design of benign apps are to meet requirements from users. However, two basic principles [12] guide the design of most malware are to (1) trigger the execution of their malicious payload (i.e., the part of malware carrying malicious behaviors) frequently to seek maximal benefits; (2) evade detection to prolong their lifetime. Guided by these principles, mobile malware leverages two major features of mobile platforms as below.

Frequent occurrences of imperceptible system events. Unlike traditional software, where events typically come from standard user inputs (keyboards and mice), a large portion of behaviors in mobile apps are driven by events from the mobile system and its sensors [13]. Compared to UI-triggered events, which rely on the user to perform a specific sequence of UI interactions in a specific app, system events are much more frequently triggered. Thus, malware often leverages system events to increase the chances of invoking its malicious payloads [4]. Moreover, system events can occur when the user is not using the app or the device itself, malicious behaviors triggered by system events can easily evade the user's attentions, concealing the signs and traces of the malicious behaviors.

Informative external-environment states. Mobile apps can access numerous attributes of the external environment (e.g., locations and system time). These attributes often convey useful information about the current states of the environment. Such environment states are frequently exploited by malware to actively control the execution of malicious behaviors. For example, the DroidDream [14] malware family suppresses its malicious behaviors during the day and invokes its malicious payload only at night. Since app reviewers or automated tools, such as Bouncer [1], can analyze apps for only a short period of time and with limited variations of environmental conditions, it is very likely that the reviewers and the tools cannot detect the malware when the environmental conditions that trigger the malicious behaviors are not met.

Based on the above-mentioned fundamental differences between malware and benign apps, we propose that *the context in* *which a security-sensitive behavior occurs* is a strong indicator of whether the security-sensitive behavior is malicious or benign. Malware executes its malicious payloads only under certain unique contexts to reach a balance between prolonging its life time and increasing the chance of being invoked. Such contexts are unique because a balance can be reached only when malicious behaviors are invoked frequently enough to meet the needs (e.g., a certain number of clicks per day to improve search engine rankings of a website), but not too frequently for reviewers/users or tools to notice the abnormal behaviors of the app. On the contrary, most of the contexts for benign behaviors are user interactive, and thus are exploited less frequently by malware.

Expressing contexts in mobile apps is a non-trivial task. In mobile apps, various elements could be used to describe the contexts in which security-sensitive behaviors occur. However, due to the complex event-driven nature of mobile apps, expressing the contexts using all the factors determining the invocation of security-sensitive behaviors would incur huge overhead in extracting the context information and extra burden in differentiating benign behaviors from malicious ones. Consider the example that a security-sensitive behavior can occur only when an app component enters into the lifecycle method that invokes the behavior. Android apps are component-based and each component has a lifecycle [15]. Any factor changing the component's state will determine the invocation of the lifecycle method, thus determining the invocation of the security-sensitive behavior. Since there are a large number of these factors, such as messages sent by other components, remote procedure calls by other components, UI operations of users, and system events, incorporating all these factors into the definition of context would make the analysis for extracting contexts expensive and bring noisy data in differentiating benign behaviors from malicious ones.

To express contexts concisely and yet capture the essence to reflect intentions, we propose an abstraction of the contexts. Such abstraction of the contexts should be detailed enough to reflect the intentions of security-sensitive behaviors, but not too redundant to include all the low-level detailed information about system states. Our *context* definition is based on the observation that activating conditions (e.g., events triggering the execution of payloads) and guarding conditions (e.g., environmental attributes controlling the execution of payloads) are the key elements of context information to differentiate malicious behaviors and benign behaviors. Thus, we define a *context* for a security-sensitive behavior as a tuple containing an *activation event* (the event that triggers the security-sensitive behavior), and a series of *context factors* (environmental attributes controlling the execution of the security-sensitive behavior).

Although our context abstraction reduces the burden in inferring context information, we still need to address two challenges posed by mobile apps. First, inferring activation events requires the analysis of the entry points of the app. Unlike desktop programs that have only one entry point for a program execution (i.e., the main function), a mobile app usually has multiple entry points due to its event-driven nature. Also, not all entry points of an app are triggered by external events, and some of them are triggered by inter-component communications (ICCs) [16] within the app. It is possible that the program execution path from the entry point triggered by an activation event to the invocation of a security-sensitive behavior goes through a chain of components of the app. Existing analysis can identify only the entry point of each component, and thus cannot be directly applied to infer activation events. Second, computing context factors requires the analysis of control flows from the activation events to the invocations of the security-sensitive behaviors. The ICCs in apps complicate the analysis because a conditional statement controlling the execution of ICCs may further control the security-sensitive behaviors in the target component of ICCs.

To address these challenges, we propose AppContext, an approach that statically analyzes the security-sensitive behaviors in an Android app. To extract activation events, AppContext chains all ICCs within the app and constructs an extended call graph (ECG) to infer activation events. To compute context factors, AppContext combines the control flows of all components from entry points triggered by activation events to the method calls that trigger security-sensitive behaviors in a reduced inter-procedure control flow graph (RICFG) [17], and leverages information flow analysis [18] to identify the environmental attributes that affect the control flows.

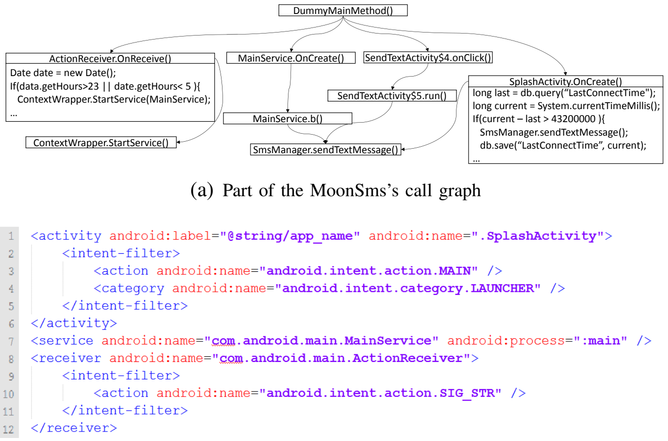
To leverage the extracted contexts for differentiating malicious behaviors and benign ones, we transform these contexts as features and use machine learning techniques, such as support vector machine (SVM) [19], to classify security-sensitive behaviors as malware or benign ones. We use machine learning techniques because the reasoning about the maliciousness of a behavior is vague and subjective by nature. Simply using a static threshold (e.g., the frequency of contexts) to differentiate malicious and benign behaviors does not perform well because it is difficult to determine a proper threshold. For many subtle cases, machine learning techniques are desirable to detect malware by taking multiple factors into account and making decisions based on rich data sets statistically.

This paper makes the following main contributions:

- An approach, AppContext, to detect malware based on the insight that the context of a security-sensitive behavior is a strong indicator of the maliciousness of the behavior.
- An abstraction to model the contexts of security-sensitive behaviors based on the two unique characteristics of malware (activation conditions and guarding conditions).
- A static-analysis technique for context extraction, which accurately identifies activation conditions and guarding conditions for security-sensitive behaviors.
- Three evaluations on 846 Android apps to demonstrate the effectiveness of AppContext.

II. Background

Android Component. Android apps are composed of four types of components. An Activity represents a user interface. A Service represents a task being processed in the background.


DummyMainMethod()
ActionReceiver.OnReceive()
MainService.OnCreate()
SendTextActivity$4.onClick()
SplashActivity.OnCreate()
SendTextActivity$5.run()
MainService.b()
ContextWrapper.StartService()
SmsManager.sendTextMessage()

```
Date date = new Date();
If(data.getHours>23 || date.getHours< 5 ){
  ContextWrapper.StartService(MainService);
...
```

```
long last = db.query("LastConnectTime");
long current = System.currentTimeMillis();
If(current – last > 43200000 ){
  SmsManager.sendTextMessage();
  db.save("LastConnectTime", current);
...
```

(a) Part of the MoonSms's call graph

```
1  <activity android:label="@string/app_name" android:name=".SplashActivity">
2      <intent-filter>
3          <action android:name="android.intent.action.MAIN" />
4          <category android:name="android.intent.category.LAUNCHER" />
5      </intent-filter>
6  </activity>
7  <service android:name="com.android.main.MainService" android:process=":main" />
8  <receiver android:name="com.android.main.ActionReceiver">
9      <intent-filter>
10         <action android:name="android.intent.action.SIG_STR" />
11     </intent-filter>
12 </receiver>
```

(b) Code snippet of MoonSms's manifest file

Fig. 1: Motivating Example in MoonSMS App

A Content Provider allows sharing of app-specific or system data across apps. A Broadcast Receiver is a component that receives broadcasted message objects, named as Intents.

Apps specify which content provider they access by using Uniform Resource Identifiers (URI). Content providers may require that apps hold certain permissions to access the providers. We name the content providers that require permissions as security-sensitive content providers.

Android Permissions. Developers can restrict access to Android components by using permissions. A component protected by a permission can be accessed by only apps that have obtained that permission. To access sensitive resources in a component, an app must request corresponding permissions. Permissions are declared in the Android Manifest (AndroidManifest.xml). At installation time, Android presents the list of permissions requested by the app. Users can either allow all permissions or give up installing the app.

Intent. A common way for Android components to communicate with each other is to send Intent messages. An Intent is a message that declares a recipient (by an action string or specific component name) and optionally includes data. Intents can be event notifications sent by the operating system to apps. These Intents are triggered by system events that can be sent only by the operating system. Permissions may be used to both restrict who may receive an intent sent by an app, and restrict who may send intents to a particular app. We name Intents that require permissions to send or receive as security-sensitive Intents. Sending or receiving a security-sensitive Intent is a security-sensitive behavior.

III. A Motivating Example

To illustrate our approach, we use a simplified malware example named MoonSms. MoonSms is a repackaged app that carries both benign functionality and injected malicious DroidDream [14] payloads. The benign functionality provides a variety of festive greetings for SMS messages. Thus, it is rational that MoonSms requests the SEND_SMS permission. Figure 1 shows that *SmsManager.sendTextMessage* (i.e., an API method that uses the SEND_SMS permission) is invoked under three contexts. Each invocation of this method is a security-sensitive behavior of the app.

The first invocation of *SmsManager.sendTextMessage* occurs when the user clicks the "Send" button in an activity component named "SendTextActivity". When the "Send" button is clicked, its *onClick* event handler spawns a new thread that invokes *SmsManager.sendTextMessage*.

The second invocation of *SmsManager.sendTextMessage* occurs when the signal strength of the device changes. When the signal strength changes, the system broadcasts an Intent containing the "SIG_STR" action. MoonSms registers a broadcast receiver component named "ActionReceiver" (Lines 8-12 in Figure 1(b)) to receive this Intent. When this Intent is broadcasted, ActionReceiver is activated and its *onReceive* method begins execution. ActionReceiver's *onReceive* method starts a service component named "MainService" by invoking the *startService* API method (when the current time is between 11 pm and 5 am), which begins executing MainService's *onCreate* lifecycle method. Finally, MainService's *onCreate* method invokes another method named *b*, which calls *SmsManager.sendTextMessage*.

The third invocation of *SmsManager.sendTextMessage* occurs when MoonSms is launched. When the MoonSms is launched, its main activity component, "SplashActivity" (Lines 1-6 in Figure 1(b)), begins execution in its *onCreate* lifecycle method. SplashActivity's *onCreate* method invokes *SmsManager.sendTextMessage* when the current time is at least 12 hours after the "LastConnectTime" is saved in a database.

In the preceding example, the first invocation is not malicious because reviewers can analyze the content on the screen and confirm that the security-sensitive behavior is expected to occur. However, the second and third invocations cannot be justified by the functionality that MoonSms is expected to provide. By inspecting the behaviors, we find that the second and third invocations are malicious because these invocations send SMS to a confirmed malicious server.

This example demonstrates that the contexts of security-sensitive behaviors are essential to differentiate between benign and malicious behaviors, especially when the benign functionality provided by apps may rationalize the requested permissions, and the security-sensitive method calls allowed by the requested permissions can also be used by malicious functionality. AppContext focuses on exposing the contexts of security-sensitive behaviors. We refer back to this example in the rest of the paper to illustrate how AppContext formalizes the abstraction of contexts of security-sensitive behaviors and extracts these contexts from app binary code.

IV. Context of Security-Sensitive Behavior

In this section, we formally define the context of a security-sensitive behavior.

We consider a **security-sensitive behavior** as an invocation of a security-sensitive method under a certain context. A security-sensitive method is a method that meets at least one of the following three requirements: (1) Permission-protected methods. Some methods in the Android API require permissions to be invoked. Such methods usually access security-sensitive resources and data (the detailed list of the methods is specified in PScout [20]). (2) The methods that is either a source method or a sink method (output channel) of an information flow. An information flow consists of a source from which the security-sensitive data may originate and a sink to which the data may be sent (the detailed list of sources and sinks are specified in Susi [21]). Sources and sinks are not always protected by permission; for example, *FileOutputStream.write* is a sink method to write the data to a file but does not require Android permissions to be invoked. A permission-protected method may not be a source/sink method; for example, *ContextWrapper.setWallpaper* is protected by permission SET_WALLPAPER, but is neither a source nor a sink. (3) Reflection methods [22] and dynamic code-loading methods [23]. Resolving reflection or dynamic loading methods in static analysis is a known difficult problem with fundamental limitations [24]. For this reason, we do not attempt to resolve these methods, but rather treat them as being security sensitive. In doing so, we are being conservative, because these methods may result in invoking other security-sensitive methods. There are a few methods in the Android API allowing apps to load and invoke code at runtime that has also been leveraged by existing malware [4] (a detail list is listed on our project website [25]).

Our definition of context (Definition 4.6) includes two important characteristics that determine the invocations of security-sensitive method calls: **activation events** (Definition 4.2) and **context factors** (Definition 4.5). Such definition represents a set of essential elements for decision making in app inspection.

The activation events are the *external events that trigger the security-sensitive methods*. The external events include UI events (events triggered by interactions on apps' graphical user interfaces), SYSTEM events (events initiated by the system-state changes such as receiving SMS), and HARDWARE events (events triggered by the interactions on the device interfaces, such as pressing the HOME or BACK button). Activation events connect security-sensitive behaviors to the behaviors' "initiator" in the external environment (e.g., users or system), as the events are triggered when the external environment changes or the mobile system reaches a certain state.

To infer activation events of security-sensitive method calls, we analyze the entry points (e.g., *ActionReceiver.OnReceive()* and *SendTextActivity$4.onClick()* in Figure 1(a)) of call graph that contains the security-sensitive method calls. In an Android app, not all entry points are triggered by activation events, and some of entry points can be triggered only by inter-component communications. For example, *MainService.OnCreate()* is triggered by *startService()* in the component *ActionReceiver*. An analysis needs to trace back a chain of entry-point methods executed before the invocation of the security-sensitive methods to identify the entry points that can be used to infer activation events.

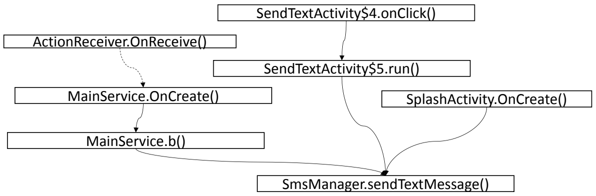

SendTextActivity$4.onClick()
ActionReceiver.OnReceive()
SendTextActivity$5.run()
MainService.OnCreate()
SplashActivity.OnCreate()
MainService.b()
SmsManager.sendTextMessage()

Fig. 2: ECG of CG shown in Figure 1(a)

To assist the analysis to locate entry points triggered by activation events, we first define an extended call graph that connects all the ICCs in an app.

Definition 4.1. An *extended call graph* $ECG = (N, E)$ for an app p is a directed graph in which each node $n \in N$ denotes a method in p, and each edge $e(a, b) \in E$ denotes either a calling relationship from a to b or a in one component A calls b in another component B. An entry point of the ECG is a node n_e that has no incoming edges (i.e., for each nodes $n \in N$, $e(n, n_e) \notin E$).

An extended call graph (ECG) is a call graph with edges representing ICCs. The entry point of ECG can be triggered by activation events. For example, Figure 2 shows part of MoonSms's ECG. Compared to the corresponding call graph (CG) shown in Figure 1(a), the ECG has an ICC edge from *ActionReceiver.OnReceive* to *MainService.OnCreate*, connecting the component *ActionReceiver* to component *MainService*. ECG enables our approach (Section V) to link the security-sensitive method call (*SmsManager.sendTextMessage*) to the entry point *ActionReceiver.OnReceive*, and the activation event (signal strength changes) can be further inferred from the entry point. We next define the activation event.

Definition 4.2. An *activation event* act_{n_e,n_k} of a method call n_k is the event that triggers the entry point n_e in an extended call graph $ECG = (N, E)$ and there exists a call path $P = n_e n_1 n_2 ... n_k$ such that $e(n_e, n_1) \in E$ and for $i = 1, 2, ..., k$, $1 \leq k$, $e(n_{i-1}, n_i) \in E$.

Activation events are identified by their *action types*, which can be inferred from entry points. Specifically, the action types of UI events are their corresponding operation types (e.g., click, long click), the action types of system events are state changes that trigger the events (e.g., signal strength changes), and the action types of hardware events are the component lifecycle phases that the events lead to (e.g., onPause, leaving the component; onResume, re-entering the component).

The context factors are *environmental attributes that control the execution of security-sensitive method calls*. The values of context factors can affect control flows from entry points triggered by activation events to security-sensitive method calls. To precisely describe the control flows in an Android app, we adopt and simplify the definition of an inter-procedure control-flow graph (ICFG) from Harrold et al. [17] and define a *reduced inter-procedure control-flow graph* (RICFG).

Definition 4.3. Given an $ICFG$, an entry point n_e, and a method call n_k, a reduced inter-procedure control-flow graph $RICFG_{n_e,n_k}$ is a subgraph of $ICFG$ that contains all the paths from n_e to n_k.

For example, Figure 3(a) shows an $RICFG_{n_e,n_k}$ where the entry point n_e is *ActionReceiver.OnReceive()* in the ECG (shown in Figure 2) and the security-sensitive method call n_k is *sendTextMessage*.

Apps usually obtain the values of the environmental attributes by using certain Java/Android API methods (e.g., *currentTimeMillis()*, *getInstalledApplications()*). We denote such API methods as *environment-property methods*. We next define control dependence among statements and use control dependence and environment-property methods to define context factors.

Definition 4.4. In a program, if a statement n_s controls whether a statement n is executed, n is *control dependent* on n_s.

Definition 4.5. Given an $RICFG_{n_e,n_k}$ and a set of conditional statements S_{n_e,n_k} in $RICFG_{n_e,n_k}$ that n_k is control dependent on, a *context factor* f_{n_e,n_k,s_i} is an environmental attribute whose value is used in a conditional statement s_i where $s_i \in S_{n_e,n_k}$.

The context factors are computed by analyzing the information flows (data dependence) from environment-property methods to conditional statements that control the execution of security-sensitive method in the RICFG. Based on these definitions, we formally define a *context*:

Definition 4.6. A *context* C_{n_e,n_k} for method call n_k is a tuple consisting of the activation event act_{n_e,n_k} and the set of context factors $\{f_{n_e,n_k,s_i} | s_i \in S_{n_e,n_k}\}$ where S_{n_e,n_k} is the set of conditional statements in $RICFG_{n_e,n_k}$.

V. AppContext

We next present AppContext, our approach that extracts the values of elements in the context definition defined in Section IV. First, AppContext constructs a call graph from an apps binary and performs static analysis to locate its security-sensitive behaviors. Next, AppContext identifies activation events by the entry points of the computed call graph, and converts the call graph into an ECG by using ICC information. Then, AppContext constructs RICFGs for each security-sensitive method calls in the ECG and traverses each RICFG to find conditional statement sets. Next, AppContext finds context factors whose values are used in conditional statements via information flow analysis and then generates the complete contexts using identified activation events and context factors. Finally, AppContext classifies the security-sensitive behaviors by using the features of the extracted contexts.

A. Locating Security-Sensitive Behaviors

AppContext locates security-sensitive method behaviors by constructing call graphs and locating security-sensitive method calls within the call graphs (we leverage Flowdroid's call graph building [26]; please check their paper [18] for details). Security-sensitive method calls are divided into three groups by the information used to identify them, as illustrated below.

First, the permission-protected API methods, source or sink methods, reflection methods, and dynamic code-loading methods are all identified by using a method signature. If a method matches a method signature in this group, AppContext extracts and saves the method name, permission, and the entry points for later analysis.

Second, the methods that read or write security-sensitive Content Providers are identified by the URIs of the content providers. To access a content provider, the URI designating the recipient content provider is passed to a *ContentResolver* class (Section II). Only the method calls using the URIs of security-sensitive content providers are security sensitive. The list of URIs designating security-sensitive content providers is provided in PScout [20]. If the URI parameter of a method is in the URI list, AppContext saves the URI, permission, and the entry points for later analysis.

Finally, the methods that send or receive security-sensitive Intents are identified by the Intent-action strings. An app can call *sendBroadcast* or *registerReceiver* with Intent action strings to send or receive specified Intent messages. The list of Intent-action strings requiring permissions to send or receive is provided in PScout [20]. If the intent parameter in the method is in the list, AppContext saves the Intent-action string, permission, and the entry points for later analysis.

B. Identifying Activation Events

As discussed in Section IV, the activation events are represented by their action types. Action types can be extracted from the app's entry points. AppContext identifies activation events by analyzing two types of entry points. (1) For system events handled by intent filters and hardware events, their entry points are *lifecycle methods*. If the components of the lifecycle methods have intent filters specified for system Intent messages, the entry points are invoked by system events. Otherwise, the entry points are invoked by hardware events. (2) For both system events captured by event-handling methods and UI events, their entry points should be *event-handling methods*.

Algorithm 1 presents the analysis used to extract activation events for the given security-sensitive method calls. The analysis returns a list of activation events ($\mathcal{E}$) for each security-sensitive method call. The analysis takes security-sensitive method calls and their corresponding entry points as input. An entry point belongs to one of two above-mentioned categories: lifecycle methods and event-handling methods.

For the first category of entry points, lifecycle methods, the analysis first decides whether the activation event could be a system event captured by intent filters (Line 6). If the component that the lifecycle method belongs to has intent filters, for each intent filter, the attributes in the intent filters are used to represent the activation events of the contexts. For each activation event, AppContext create a tuple and saves activation event along with the method call and the entry point in the tuple to the $\mathcal{E}$ list for later analysis (Line 9).

The analysis then decides whether the lifecycle method can be invoked by ICC calls (e.g., *startService*, *sendBroadcast*) (Line 13). If there are method calls invoking the lifecycle method, the analysis adds ICC edges to the CG (Line 14), and replaces entry points of the ICC calls with the original entry

Algorithm 1: IdentifyActivationEvent

```
Inputs  : $\mathcal{B}$: A set of contexts without context factors and activation events (i.e.,
          tuples consisting of security-sensitive method calls and their entry
          points in call graphs)
          $CG$: The call graph of the whole app
          $\mathcal{A}$: App binary code
Outputs: $\mathcal{E}$: A set of contexts without context factors (i.e., tuples consisting of
          security-sensitive method calls, their activation events, and
          corresponding entry points)
          $ECG$: The extended call graph of the whole app
1  begin
2      $\mathcal{E} \leftarrow \emptyset$
3      foreach $b \in \mathcal{B}$ do
4          $entrypoint \leftarrow getEntrypoint(b)$
5          if $isLifeCycleMethods(entrypoint)$ then
6              if $hasIntentFilters(entrypoint, A)$ then
                   // System events (by intent filters)
7                  $Filters \leftarrow getFilters(entrypoint, A)$
8                  foreach $filter \in Filters$ do
9                      $\mathcal{E}.addFilter(b, filter)$
10                 end
11             end
12             $ICC \leftarrow findICCs(CG, entrypoint)$
13             if $ICC \neq \emptyset$ then
                   // adding ICC edges
14                 $CG.add(ICC)$
                   // Recursively invoke the algorithm
15                 $E \leftarrow replaceEntryPoint(b, CG)$
16                 $\mathcal{E}.addAll(identifyActivationEvent(E, CG, A))$
17             end
18             else
                   // Hardware events
19                 $\mathcal{E}.addLifeCycle(c)$
20             end
21         end
22         if $isEventHandler(entrypoint)$ then
23             $\mathcal{E}.addHandler(c)$
24         end
25     end
26     $ECG \leftarrow CG$
27     return $\mathcal{E}$
28 end
```

points (Line 15). Then Algorithm 1 is invoked recursively with the augmented CG (i.e., ECG) and new entry points to cover all activation events. The activation events are then saved in the tuples for later analysis (Line 16).

If the lifecycle method cannot be invoked from app code, then the security-sensitive method call is triggered by hardware events. We use the attributes of the lifecycle methods to represent the activation events, and save the activation events in the tuples for later analysis (Line 19).

For the second category of entry points, event-handling methods, the analysis uses the attributes of the UI event-handling methods or system event-handling methods to represent the activation events, and save the activation events in the tuples for later analysis (Line 23).

C. Extracting Context Factors

After computing the ECG and activation events for a security-sensitive method call, AppContext constructs and traverses the RICFGs to extract context factors. As shown in Section IV, the RICFGs need to be constructed based on the ECG. Thus, for each security-sensitive method call, AppContext identifies the ECG's entry points that can lead to the invocation of the method. Then AppContext obtains the ICFG of the app by connecting the CFG of each node on the ECG. Based on the ICFG, AppContext constructs the RICFGs from each entry point to the security-sensitive method call. For each RICFG, AppContext traverses the RICFG to identify the conditional statements on which the security-sensitive method is control-dependent. Finally, AppContext saves the set of extracted conditional statements with the security-sensitive method call and the corresponding activation events.

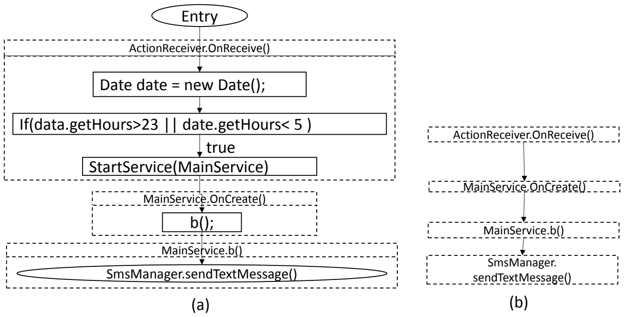

Entry
ActionReceiver.OnReceive()
Date date = new Date();
If(data.getHours>23 || date.getHours< 5)
true
StartService(MainService)
MainService.OnCreate()
b();
MainService.b()
SmsManager.sendTextMessage()
(a)
ActionReceiver.OnReceive()
MainService.OnCreate()
MainService.b()
SmsManager.
sendTextMessage()
(b)

Fig. 3: An RICFG (a) and its corresponding ECG subgraph (b)

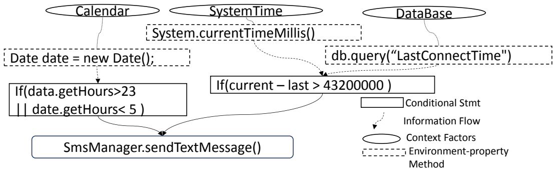


Fig. 4: Context factors of MoonSms in Figure 1

Figure 4 presents the analysis used to extract context factors. For each conditional statement extracted in the previous step, AppContext tracks the information flow from the environment-property methods (Section IV) to the conditional statement. The sources of the information flows indicate which context factors control the invocation of the security-sensitive behaviors. In the MoonSms example, the context factors are Calendar information, system time, and database information. By combining the context factors with corresponding activation events of the security-sensitive method calls, AppContext generates the complete context tuples.

D. Classifying Security-Sensitive Behaviors

Leveraging the extracted contexts to classify security-sensitive behaviors as malicious and benign, we formulate the detection of malicious behaviors as a classification problem. AppContext leverages a supervised learning approach to train a classifier to compute the conditional likelihood of a security-sensitive behavior being malicious versus benign given context features. Specifically, AppContext uses a support vector machine (SVM) as the classifier because SVM is very resilient to over-fitting even with a large number of values.

Classification is performed using a set of features. A feature is a function that associates a training example with a value, i.e., a function evaluates a certain single domain-specific criterion for the example. AppContext leverages the list of features in Table I for classifying security-sensitive behaviors. The list consists of the features about the security-sensitive behavior itself, and the features describing the contexts of the behavior: the activation events and the context factors. With

TABLE I: FEATURE CATEGORIES FOR CLASSIFICATION

Features of Behavior Information		
Permission	Security-sensitive method call	
Features of Activation Event		
Hardware event	System event	UI event
Features of Context Factors		
List of environmental attributes		

this list of features, AppContext generates a feature vector for each context of a security-sensitive behavior.

Table II shows an example of feature vectors. For features describing behavior information (i.e., Permission, Method Call), the feature values are the name of the permission or method. For methods such as source/sink, reflection, or dynamic loading methods that do not have corresponding permissions (i.e., do not require permissions to be invoked), the permission names are predefined strings such as "SOURCE", "SINK", "REFLECTION", "DYNLOADING". For features describing activation events, the feature values are the action types (Section IV) of the events. For features describing the context factors ($F_1, F_2, ..., F_{142}$), the feature values are either "1" (the context contains the context factor) or "0" (the context factor is not part of the context).

VI. TOOL IMPLEMENTATION

In this section, we briefly illustrate implementation details of AppContext. More information can be found on our project website [25].

Static Analysis. AppContext leverages Soot [27] as its underlying static analysis framework. AppContext uses Dexpler [28] to convert Dalvik bytecode into the Jimple intermediate representation from which Soot constructs its call graph. AppContext also leverages FlowDroid [18], a static taint analysis tool based on Soot, to provide a precise modeling of the Android component lifecycles and callback‘ methods.

Context extraction. To extract contexts, AppContext uses the permission mappings provided by PScout [20] as input and performs the analysis discussed in Section V. Since AppContext relies on PScout's mappings, the soundness and completeness of the mappings may affect the number of false positives and false negatives produced by AppContext.

VII. EMPIRICAL EVALUATIONS

To evaluate the effectiveness of AppContext and using context information to detect malware, we have conducted three evaluations. We seek to answer the following research questions:

- **RQ1**: How effective is AppContext in identifying malware? How does AppContext compare to the approach without context information in terms of the effectiveness of malware identification?
- **RQ2**: How do activation events and context factors in our context definition contribute to the effectiveness of malware identification?
- **RQ3**: How accurate is our static analysis in inferring contexts?

A. Study Subjects

Our subject apps include 846 Android apps in total (633 benign apps, 202 malicious apps, and 11 open-source apps). To collect malicious apps, we randomly select 130 malicious apps from a malware dataset collected by Zhou et al. [4], 30 malicious apps from the VirusShare dataset [29], and 50 malicious apps from the Contagio dataset [30]. We also select 17 malicious apps identified by VirusTotal [31] that were posted on Google Play in 2013 but were later removed by Google. Our final malware dataset contains 202 malicious apps. These malicious apps cover the majority of existing Android malware families from 2011 to 2014, which are rapidly evolving to circumvent detection by various mobile security software.

To collect benign apps, we download the top 500 apps for each category from Google Play as of January 2013. Because FlowDroid runs out of memory on large apps, to ensure that enough apps can be analyzed without errors, for each category, we randomly select 20 apps under 5 MB and 20 apps without size restriction from these top 500 apps. We also exclude the apps identified as malware by VirusTotal and the apps that cause FlowDroid to throw exceptions or timeout. The final benign dataset contains 633 apps. To collect open-source apps, we randomly select 15 apps from F-Droid [32]. Among these 15 apps, we exclude 4 apps that do not have security-sensitive behaviors. Our open-source dataset contains 11 apps.

We apply AppContext to extract contexts from the subject apps. AppContext runs on a desktop with 3.4 GHz Intel Core i7 processor and 8 GB of memory. We set the timeout of AppContext as 80 minutes, and AppContext exceeds the timeout limit for 162 apps, which are then excluded from the later study. For the 846 apps being used as subjects, AppContext takes on average 647 seconds to finish the analysis of one app.

B. RQ 1: Overall Effectiveness

To answer RQ1, we label the extracted contexts from the subject apps, and perform a ten-fold cross-validation to evaluate the overall effectiveness of AppContext. To make a fair comparison with the existing approaches that do not use context information, we apply the supervised learning approach using all the features of AppContext, and then apply the same supervised learning approach using the features containing only the behavior information shown in Table I (i.e., security-sensitive method calls and permissions). The results are shown in Table III and Table IV, respectively (the second and third rows).

Labelling security-sensitive method calls. Because there is no ground truth for determining a security-sensitive method call as malicious or benign, as a best-effort solution, we systematically label security-sensitive method calls as malicious based on the existing malware signatures [31], [33], [34]. Specifically, we label a security-sensitive method call as malicious if the class/package name of the method call matches any class/package name that we collected from the

TABLE II: FEATURE VECTORS FOR MOONSMS EXAMPLE

Permission	Method Call	Hardware	System	UI	F_1	F_2	F_3*	F_4*	F_5*	F_6	...	F_{142}
SEND_SMS	*sendTextMessage*	N/A	SIG_STR	N/A	0	0	1	0	0	0	...	0
SEND_SMS	*sendTextMessage*	EnterApp	N/A	N/A	0	0	0	1	1	0	...	0
SEND_SMS	*sendTextMessage*	N/A	N/A	Click	0	0	0	0	0	0	...	0

* F_3 = Calendar, F_4 = System Time, F_5 = Database

TABLE III: MALICIOUS SECURITY-SENSITIVE BEHAVIORS IDENTIFIED BY APPCONTEXT

Features Used	P(%)	R(%)
Complete Context (C)	94.8	84.8
Behavior Information (B)	79.0	37.3
Activation Events (E)	83.2	49.5
Context Factors (F)	90.6	71.2
B & E	88.0	71.3
B & F	90.2	76.9
E & F	92.5	77.3

TABLE IV: IDENTIFICATION OF MALWARE BY APPCONTEXT

Features Used	TP	FP	FN	P(%)	R(%)
Complete Context (C)	192	27	10	87.7	95.0
Behavior Information (B)	169	78	33	68.4	83.6
Activation Events (E)	163	78	39	67.6	80.6
Context Factors (F)	150	26	52	85.2	74.2
B & E	193	63	9	75.3	95.5
B & F	180	46	22	79.6	89.1
E & F	187	27	15	87.3	92.5

TP = True Positive, FP = False Positive, FN = False Negative
P = Precision, R = Recall

existing malware signatures. We label the rest of security-sensitive method calls as benign.

We collect class/package names from malware signatures of three sources. (1) Apposcopy [33] includes a list of semantic signatures for existing malware along with a tool to check apps' binaries against the signatures. We run all of the subject apps using a tool that we reproduced based on Apposcopy and record the names of the packages and classes that match the signatures. (2) We use class names in Androguard's signature database [34]. (3) The VirusTotal [31] service inspects malware by using a number of antivirus software and reports the family that the malware belongs to. We identify the malware family that each of our malicious apps belongs to using VirusTotal, and we identify the package/class names of the malicious payloads from the online technical reports provided by the antivirus software vendors for each malware family.

Cross Validation. We use the labeled behaviors (i.e., method calls) both as training and test data in a ten-fold cross-validation [35], which is a standard approach for evaluating machine-learning techniques. It works by randomly dividing all data into 10 equally sized buckets, training the classifier on 9 of the buckets, and classifying the remaining bucket for testing. This process is repeated 10 times, with each of the 10 buckets used exactly once as the testing data. We report the average precision and recall in Table IV.

Results. We evaluate the effectiveness of AppContext in identifying both malicious behaviors and malicious apps. An app is identified as a malicious app if any of its security-sensitive method calls is identified as malicious. Table III and Table IV show that AppContext (the row of Complete Context) has higher precision and recall in both identifying malicious behaviors and identifying malware than the existing approach that does not use context information (the row of Behavior Information). We next present two major reasons that cause such misidentification.

First, AppContext misidentifies a number of security-sensitive method calls triggered by UI events and without context factors. This result suggests that compared to system events and hardware events, UI events have less indication of the maliciousness of a security-sensitive method call.

Second, a few method calls are incorrectly identified as malicious because we mistakenly label similar benign behaviors as malicious. In malicious payloads, a small number of security-sensitive method calls may not have malicious intentions, such as *MediaPlayer.pause* protected by the WAKE_LOCK permission in malicious payloads. However, as we label all security-sensitive method calls in a malicious payload, AppContext incorrectly identifies such benign method calls as malicious. This result suggests that the identification results can be improved if the training set for the classifier is labeled more accurately.

We also evaluate the effectiveness of AppContext in identifying malicious reflective calls or dynamic code-loading method calls. AppContext shows high precisions and recalls in identifying malicious method calls. AppContext correctly identifies 872 out of 922 malicious method calls but also misidentifies 180 benign method calls as malicious (i.e., 82.9% precision, 94.5% recall). AppContext correctly identifies 710 out of 787 malicious dynamic code-loading method calls but misidentifies 137 benign method calls as malicious (i.e., 83.8% precision, 90.2% recall). For all 56 malicious apps using root exploits (which are commonly launched by dynamic code loading [4]), only one malicious app (i.e., AsRoot) was not identified by AppContext. As the detailed behaviors of reflective calls and dynamically-loaded code were unobtainable in static analysis, such results show the advantage that AppContext can differentiate benign and malicious security-sensitive method calls without knowing the detailed behaviors being triggered.

C. RQ2: Effectiveness of Activation Events and Context Factors

RQ2 evaluates the effectiveness of both activation events and context factors in identifying malicious app behaviors. To answer RQ2, we use only partial features listed in Table I to train the classification model. We apply the same supervised learning approach used in RQ1 with the features being the activation events (the row of Activation Events), context factors (the row of Context Factors), behavior information and activation events (the row of B & E), behavior information and context factors (the row of B & F), and activation events and context factors (the row of E & F), respectively. The results are shown in Table III and Table IV.

Results. We evaluate the effectiveness of activation events by comparing the result of the analysis using activation events (the rows of Complete Context, B & E, and E & F) to the result of the analysis not using activation events (the rows of B & F, B, and F) in Table III and Table IV. The comparison shows that adding the features of activation events to the analysis improves both the precision and recall of the identification results. We find that the improvements are mainly because activation events help effectively identify malicious method calls that have no context factors. The activation events in some of these malicious method calls are often used by benign apps to update the UI to inform users that certain events have occurred. For example, UMS_DISCONNECTED is used to inform users that the device has been disconnected from USB mass storage, SIG_STR is used to inform users that the phone signal strength changes, and ACTION_POWER_CONNECTED is used to inform users that external power has been connected to the device. Because these events are seldom used in benign apps to trigger security-sensitive method calls, the activation events can effectively differentiate benign and malicious behaviors with no context factors.

We also evaluate the effectiveness of context factors by comparing the results of the analysis using context factors (the row of Complete Context, B & F, and E & F) with the result of analysis not using context factors (the row of B & E, B, and E). The result shows that the analysis using context factors has relatively higher precisions (over 90% for identifying malicious behaviors and around 80% for identifying malware). We find that the improvement in the precision is mainly because context factors effectively help identify the malicious behaviors whose activation events are UI events. We also find that context factors can disambiguate the malicious and benign intentions for certain vague cases when security-sensitive method calls are protected by commonly-used resources (e.g., Internet). For example, we find that some of benign apps and malware will both connect to servers (*URL.openConnection*) after the apps start, and thus the activation events and behaviors for both apps are the same. However, the context factors of malware include data from an Intent message (*Intent.getExtras*) and data from the Internet (*URL.openStream*), suggesting that whether the apps connect to the server or not is determined by whoever sends the Intent message or the Internet data. Such context factors demonstrate the command & control nature of certain families of malware.

In addition, context factors also reflect controls of security-sensitive method calls in benign apps. For example, we find that a few benign apps and malware obtain device information (*TelephonyManager.getDeviceId* etc.) after the apps start. The difference between two types of apps is that the benign apps invoke *getDeviceId* only when auto logins are successful (i.e., the context factors for *getDeviceId* include information from the database or the Internet). But malware directly sends device information to the server (i.e, no context factors).

TABLE V: Effectiveness of context extraction

# App	# Context	# Verified Context	Time(sec)
11	88	82	291

Finally, we further evaluate the effectiveness of contexts by running analysis using features of activation events and context factors (the row of E & F). The precision and recall of the analysis are comparable as the precision and recall of the analysis using complete context. Such results suggest that contexts can identify a number of malicious method calls without knowing the detailed behaviors being triggered, consistent with the analysis result for behaviors that invoke reflection or dynamic code-loading methods. Both results indicate that the maliciousness of a security-sensitive method call is more closely related to the behavior's intention (reflected via contexts) than the type of the security-sensitive resources that the behavior accesses.

D. RQ3: Accuracy of Static Analysis

To evaluate the effectiveness of the extracted contexts, we dynamically verify whether the security-sensitive method is invoked by triggering the activation events and configuring context factors based on the contexts. The execution path triggered by the activation events may vary when the context factors are assigned different values. In this evaluation, we use only open-source apps as the subjects. The main reason is that these apps come with source code, which can be used to easily infer the correct values of context factors in controlling the execution of the security-sensitive method calls. AppContext is applied on 11 open-source apps to extract contexts and the analysis time is logged.

To verify the correctness of context factors, we analyze the source code to check whether a security-sensitive method call is control dependent on each context factor. If the control dependence exists, we determine the values of the context factors that lead to the execution of the security-sensitive method call. We then configure the external environment based on the inferred values of context factors and trigger the activation events of 88 security-sensitive behaviors of these apps.

We use the activity manager through the Android Debug Bridge (ADB) to simulate system events, and manually simulate hardware and UI events. We configure the values of each context factor by changing configuration of emulators. Then, we use the profiler of the activity manager to log the executions of the apps. To monitor the execution traces, we start the profiler before firing the activation events and stop the profiler 5 seconds afterwards.

The preceding evaluation process has some limitations. The profiler cannot trace the invocations of the *onCreate* or *onDestoy* methods, because the profiling must be started after the creation of an app's process and be stopped before the destruction of the app's process. We also exclude events that cannot be simulated through ADB such as error events (e.g., triggering the *onError* method in *MediaPlayer.OnErrorListener*) and the context factors whose value we cannot manipulate such as data from URL connection).

Results. Table V shows our evaluation results. Among the 88 generated contexts, we are able to confirm 82 contexts (i.e., 93.2% accuracy). Six contexts cannot be verified because the activation events could not trigger the security-sensitive method calls. The context factors of all the contexts whose activation events could trigger the security-sensitive method calls are accurate. The average analysis time is 291 seconds, which is acceptable for the app reviewing process. Note that the evaluation result is conservative since the inferred values for context factors may not be accurate.

VIII. Threat to Validity

Threats to External Validity. Due to the current limitation in our implementation and testing environment, our dataset consisted of randomly selected apps that were smaller than 5MB, which may not be representative of the entire market. We plan to address this limitation in the future and include market apps whose sizes are larger than 5MB to further reduce the threats. The subjects from malware datasets can be biased by their selection methodologies [4], we chose our malware set from different sources to alleviate the bias in the subject selection.

Threats to Internal Validity. Inaccuracies in labeling behaviors are inevitable due to the lack of ground truth for identifying malicious behaviors. In addition, there may be human errors in collecting statistics and studying the evaluation results. These threats are mitigated by double-checking all manual work and ensuring that the results were agreed upon by at least two authors.

IX. Related Work

Contexts of Permission Uses. Pegasus [36] constructs permission event graphs using static analysis to model the effects of the event system and API semantics, and performs model checking to enforce the policies specified by users. However, specifying these policies requires that users have established knowledge about the expected behavior/functionality of the app and an understanding of the Android platform. Our approach complements Pegasus by providing the contexts, which can be used to construct Pegasus' policies. AppIntent [9] presents a sequence of GUI events that lead to data transmissions and let analysts decide whether the data transmissions are intended. AppIntent handles only app behaviors activated by GUI events while our approach analyzes a more comprehensive set of contexts (e.g., receivers and background services) and can complement their approach to handle data transmissions that are not triggered by sequences of GUI manipulations. AsDroid [37] detects stealthy app behaviors by identifying mismatches between API invocations and the text displayed in the GUIs. Our approach focuses on the events that trigger app behaviors rather than the textual analysis of the GUIs. Since app behaviors can occur without displaying a GUI, the textual analysis of GUIs alone is insufficient to detect all stealthy app behaviors. DroidAPIMiner [38] identifies malicious apps by performing frequency analysis of API invocations within a set of benign and malicious apps to extract the features of malware, and uses machine learning to determine the most relevant features. Our approach focuses on what causes security-sensitive API calls to be used rather than the pattern of API calls that are used. WHYPER [39] examines whether app descriptions provide any justification for the app's permission uses. WHYPER focuses on *why* apps request permissions while our approach focuses on *how* apps actually use the requested permissions.

Risk Ranking and Certification of Apps. Peng et al. [40] present the risk information of an app compared to other apps by using probabilistic generative models to calculate risk scoring of the app. MAST [41] triages Android apps by analyzing features extracted from the APKs. MAST uses machine learning techniques to measure the correlation between features and directs malware analysis resources to the apps that have the greater potential of risks. Kirin [42] performs lightweight certification of apps by identifying dangerous app configurations against a set of security rules. These approaches leverage various kinds of features or configurations in apps to identify potential risks. Unlike these approaches that present the risk scores or ranking for users, our approach analyzes the bytecode of apps to extract the contexts of permission uses. However, our approach can complement risk ranking and certification techniques by providing the extracted permission contexts as another metric for their evaluation.

Malware Detection. Our approach complements existing malware-detection analysis by identifying contexts that indicates the intentions of data uses. There are various approaches that perform analysis to detect malicious behaviors, such as dynamic taint analysis [7], [43], language-based information flow [44]–[47], static analysis [33], [48]–[50], and Bayesian classification [51]. However, these approaches are concerned about *how* privacy-sensitive data protected by permissions are used, while our approach provides the *contexts* under which the permissions are triggered.

X. Conclusion

We have presented AppContext, an approach based on an abstraction that extracts and transforms the context information into a set of essential elements to differentiate benign and malicious behaviors. In our evaluations, AppContext correctly identifies 192 out of 202 malicious apps with 87.7% precision and 95% recall. Our evaluation results suggest that the maliciousness of a security-sensitive behavior is more closely related to the behavior's intention (reflected via contexts) than the type of the security-sensitive resources that the behavior accesses.

Acknowledgment

This material is based upon work supported by the Maryland Procurement Office under Contract No. H98230-14-C-0141. This work is also supported in part by NSF grants CNS-1253346, CNS-1222680, CNS-1434582, CNS-1439481, CCF-1349666, CCF-1409423, CCF-1434590, and CCF-1434596, and a Google Faculty Research Award.

References

[1] G. M. Blog, "Android and security." [Online]. Available: http://googlemobile.blogspot.com/2012/02/android-and-security.html

[2] "App review — Apple App Store." [Online]. Available: https://developer.apple.com/app-store/review/

[3] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day Android malware detection," in *Proc. MobiSys*, 2012.

[4] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. S&P*, 2012.

[5] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," in *Proc. NDSS*, 2012.

[6] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proc. NDSS*, 2012.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. OSDI*, 2010.

[8] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Proc. FASE*, 2013.

[9] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection," in *Proc. CCS*, 2013.

[10] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proc. CCS*, 2013.

[11] M. Hypponen, "Malware goes mobile," *Scientific American*, no. 5, 2006.

[12] B. B. Rad, M. Masrom, and S. Ibrahim, "Camouflage in malware: From encryption to metamorphism," *IJCSNS*, 2012.

[13] "Android sensors overview." [Online]. Available: http://developer.android.com/guide/topics/sensors/sensors_overview.html

[14] T. Bradley, "DroidDream becomes Android market nightmare." [Online]. Available: http://www.pcworld.com/businesscenter/article/221247/droiddream_becomes_android_market_nightmare.html

[15] D. Franke, C. Elsemann, S. Kowalewski, and C. Weise, "Reverse engineering of mobile application lifecycles," in *Proc. WCRE*, 2011.

[16] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. MobiSys*, 2011.

[17] M. J. Harrold, G. Rothermel, and S. Sinha, "Computation of interprocedural control dependence," in *ACM SIGSOFT Software Engineering Notes*, no. 2, 1998.

[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. PLDI*, 2014.

[19] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, no. 3, 1999.

[20] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proc. CCS*, 2012.

[21] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing Android sources and sinks," in *Proc. NDSS*, 2014.

[22] S. Chiba, "Load-time structural reflection in Java," in *Proc. ECOOP*, 2000.

[23] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in Android applications," in *Proc. NDSS*, 2014.

[24] B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for Java," in *Proc. APLAS*, 2005.

[25] "AppContext," https://sites.google.com/site/asergrp/projects/appcontext/.

[26] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using Spark," in *Proc. CC*, 2003.

[27] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *Proc. CASCON*, 1999.

[28] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot," in *Proc. SOAP*, 2012.

[29] "VirusShare." [Online]. Available: https://www.virusshare.com

[30] "Contagio mobile - mobile marewale mini dump." [Online]. Available: http://contagiominidump.blogspot.com/

[31] "VirusTotal - free online virus, malware and URL scanner." [Online]. Available: https://www.virustotal.com/en/

[32] F-Droid, "FOSS apps for Android." [Online]. Available: https://f-droid.org/

[33] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. FSE*, 2014.

[34] "Androguard." [Online]. Available: https://code.google.com/p/androguard/wiki/databaseandroidmalwares

[35] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *IJCAI*, no. 2, 1995.

[36] K. Chen, N. Johnson, V. D'Silva, K. MacNamara, T. Magrino, E. Wu, M. Rinard, and D. Song, "Contextual policy enforcement for Android applications with permission event graphs," in *Proc. NDSS*, 2013.

[37] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction," in *Proc. ICSE*, 2014.

[38] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in Android," in *Proc. SecureComm*, 2013.

[39] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proc. USENIX Security*, 2013.

[40] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of Android apps," in *Proc. CCS*, 2012.

[41] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "MAST: Triage for market-scale mobile malware analysis," in *Proc. WiSec*, 2013.

[42] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. CCS*, 2009.

[43] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser: Protecting sensitive data leaks using application-level taint tracking," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, 2011.

[44] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J.Sel. A. Commun.*, vol. 21, no. 1, 2006.

[45] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, 2000.

[46] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Pro. POPL*, 1999.

[47] I. Roy, D. E. Porter, M. D. Bond, K. S. Mckinley, and E. Witchel, "Laminar: Practical fine-grained decentralized information flow control," in *Proc. PLDI*, 2009.

[48] X. Xiao, N. Tillmann, M. Fahndrich, J. De Halleux, and M. Moskal, "User-aware privacy control via extended static-information-flow analysis," in *Proc. ASE*, 2012.

[49] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proc. CCS*, 2014.

[50] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. CCS*, 2014.

[51] O. Tripp and J. Rubin, "A Bayesian approach to privacy enforcement in smartphones," in *Proc. USENIX Security*, 2014.